Estruturas de Dados – Aula 11

Prof. Dr. Eduardo Takeo Ueda
eduardo.tueda@sp.senac.br

Fila de prioridade

Fila de prioridade é uma estrutura de dados que mantém uma coleção de elementos, cada um com uma prioridade.

As operações básicas sobre uma fila de prioridade são:

- Inserir um novo elemento na fila de prioridade;
- Remover o elemento com maior prioridade da fila de prioridade.

Fila de prioridade com lista

- Não é difícil implementar uma fila de prioridade com uma lista.

- Por exemplo, podemos ter uma lista ordenada:
  - na inserção a fila é percorrida até o ponto de inserção;
  - a remoção sempre ocorre no primeiro elemento da fila.

- Mas qual a eficiência?
  - $O(n)$ para inserção;
  - $O(1)$ para remoção.

- Se a lista fosse não ordenada teríamos $O(1)$ para inserção e $O(n)$ para remoção.

Fila de prioridade com heap

Como implementar uma fila de prioridade de tal forma que tanto a inserção quanto a remoção sejam eficientes?

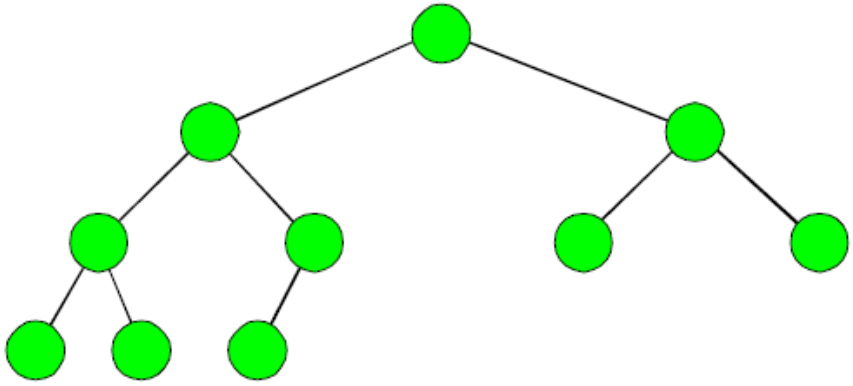
Resposta:

Podemos utilizar uma estrutura de dados chamada heap.

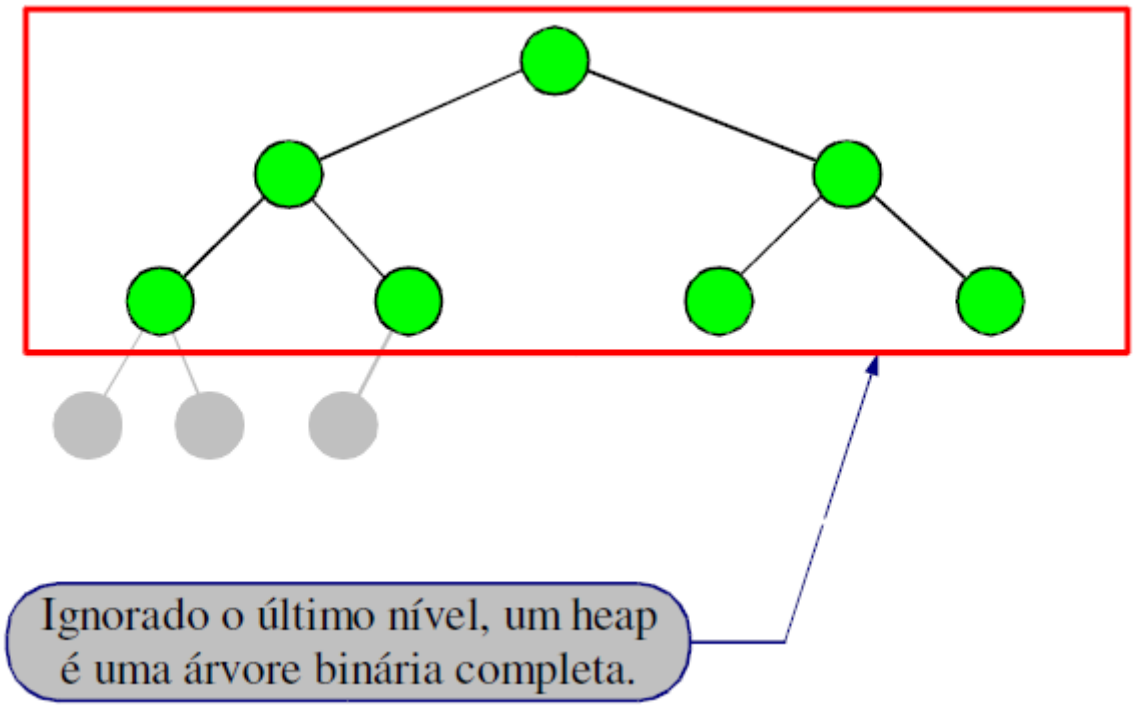
Fila de prioridade com heap

- Heap é uma estrutura de árvore binária em que cada nó não folha tem prioridade maior ou igual à prioridade de seus nós filhos.

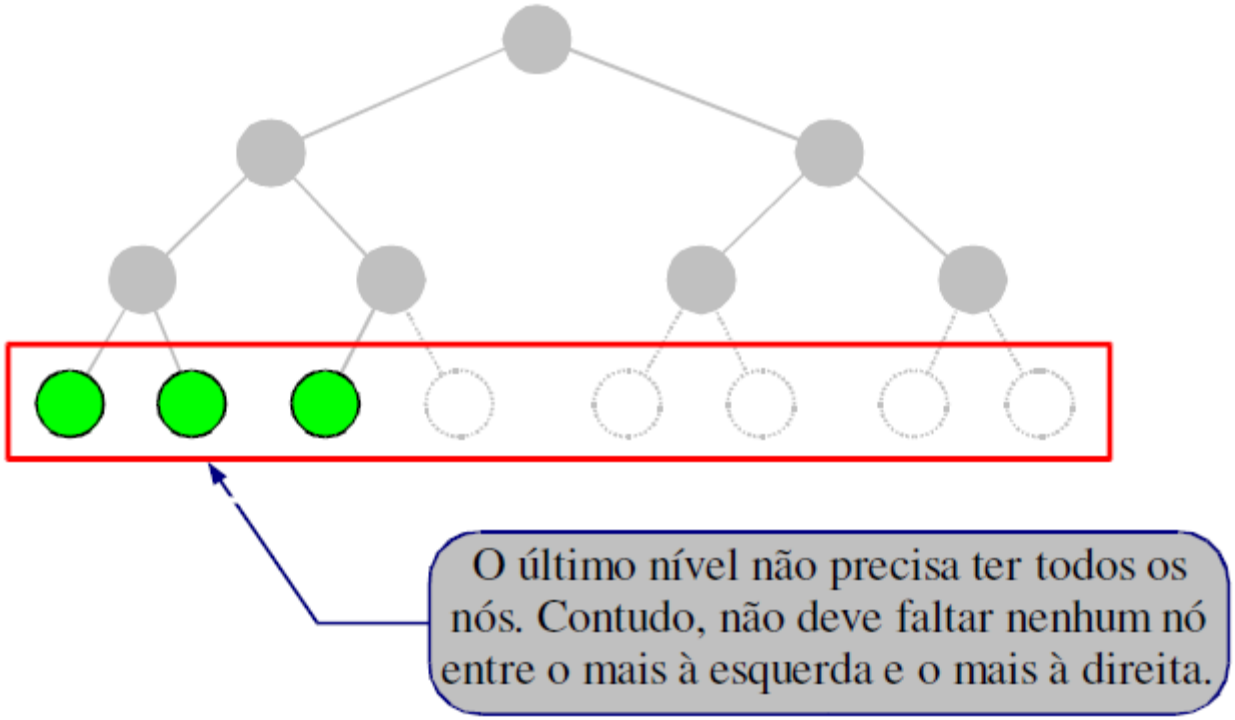
- O heap é uma árvore quase completa, ou seja, é uma árvore binária balanceada.



Estrutura heap

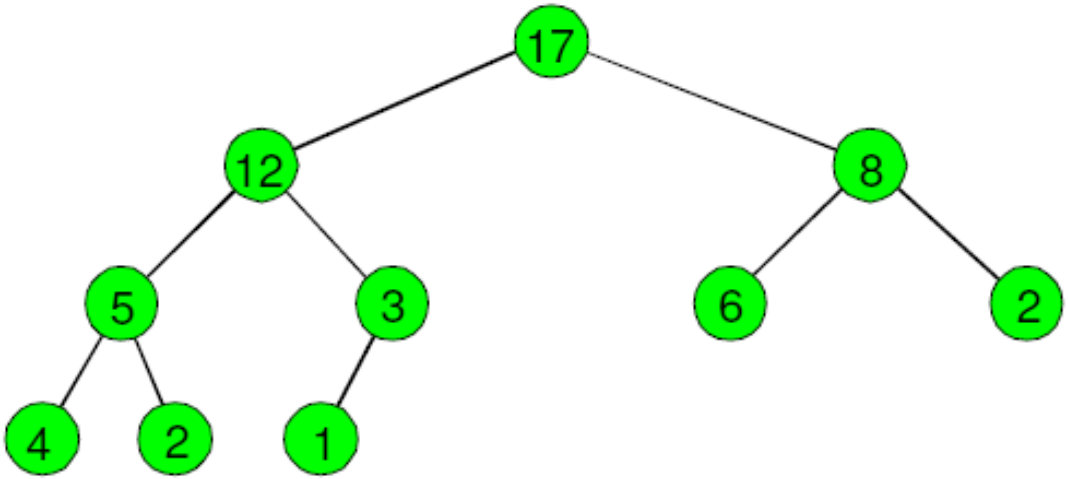


Estrutura heap



Estrutura heap

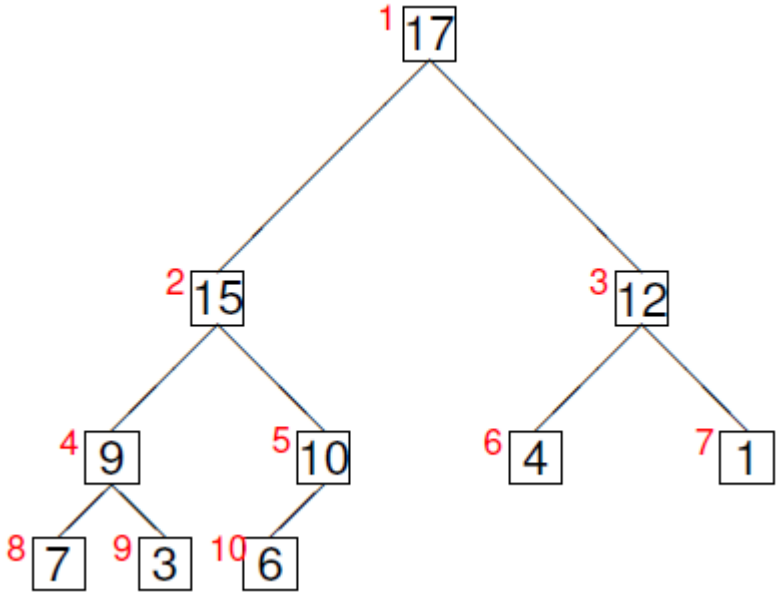
- Cuidado para não confundir heap com árvore binária de busca.



- Em um heap de máximo (mínimo) o maior (menor) elemento está na raiz da árvore.

Relação entre pai e filhos no heap

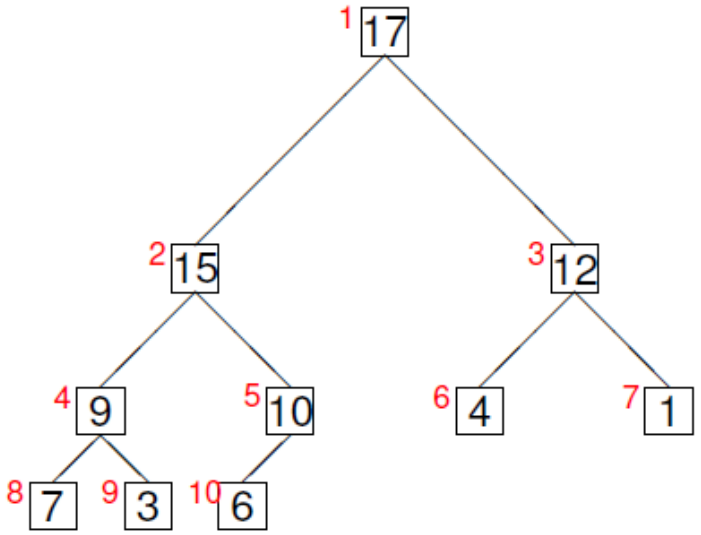
- O pai do nó i é o nó i div 2 (isto é, quociente inteiro de i / 2).
- Se o nó i tem um filho esquerdo, este será o nó 2i; se i também tem um filho da direita, este será o nó 2i + 1.



Implementação do heap em vetor

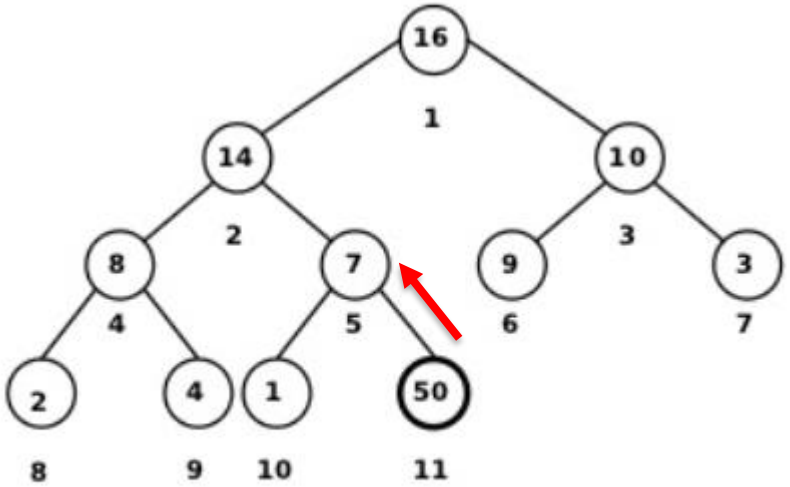
- Podemos usar um vetor H (sequencial) para representar um heap.

- A estrutura de árvore binária está implícita nas posições dos nós.

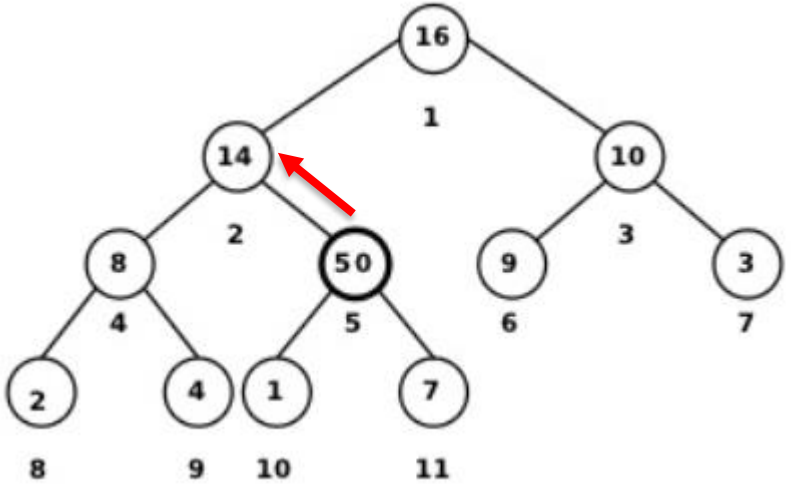


	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
H	17	15	12	9	10	4	1	7	3	6				

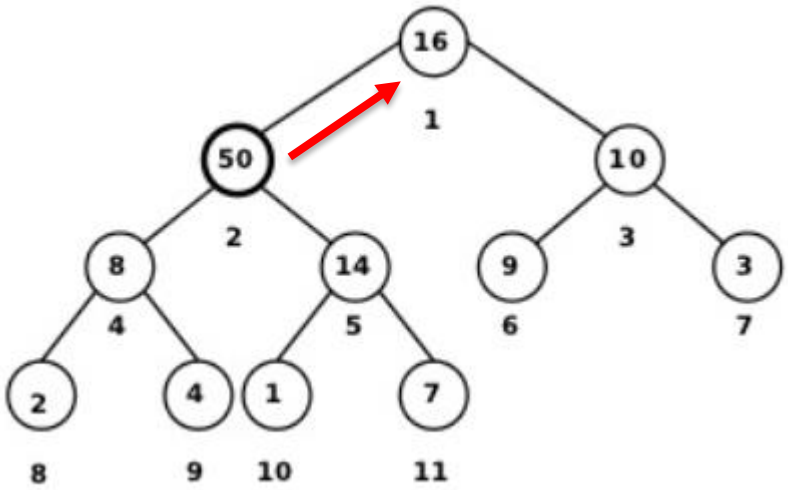
Inserção no heap

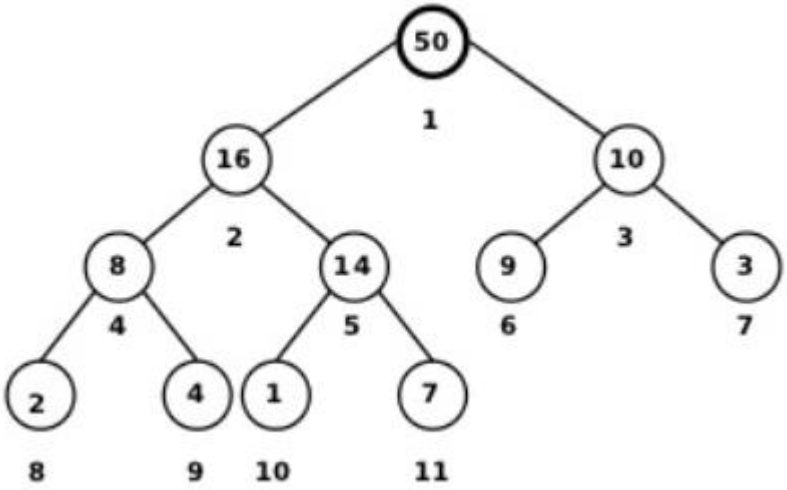


Inserimos o nó sempre no último nível(final do heap).

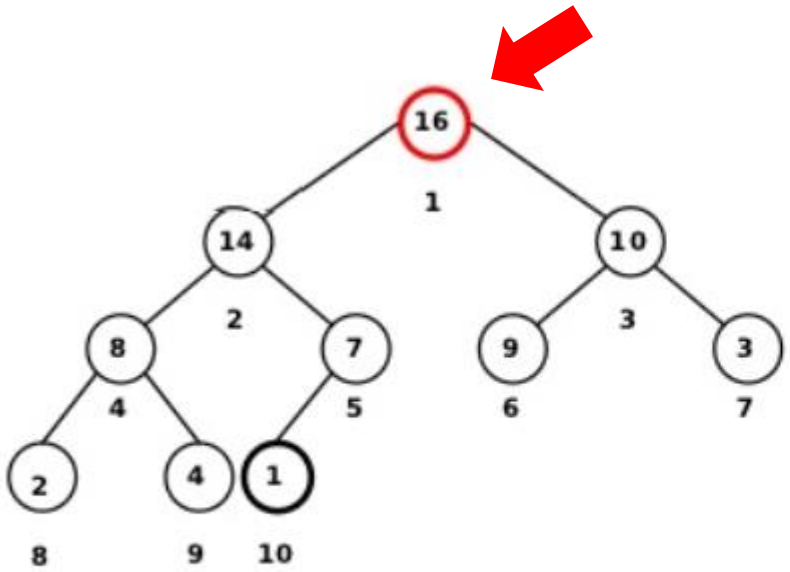


Inserção no heap

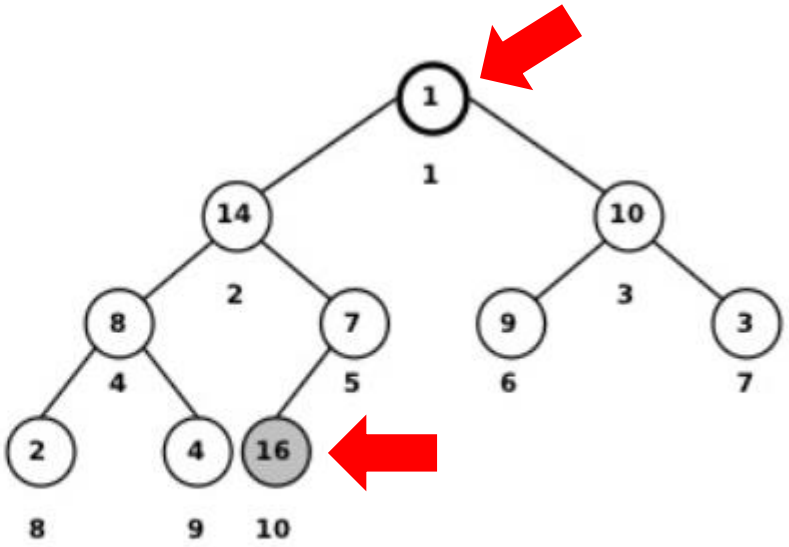




Remoção do heap

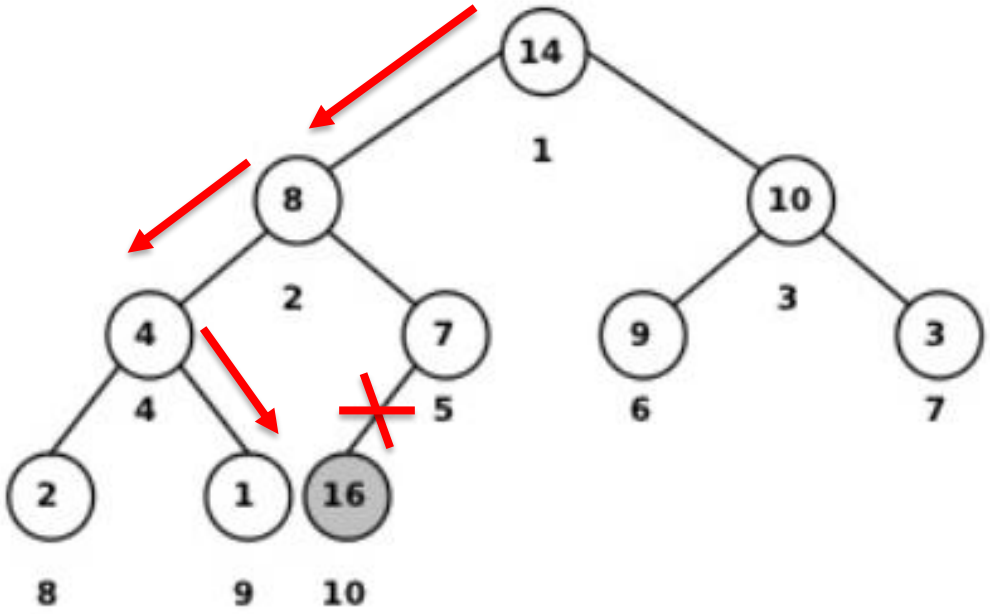


Retira-se sempre o 1º nó



Troca o 1º nó com o último

Remoção do heap



Implementação do heap

A maneira padrão de implementar um heap é utilizando um vetor.

```
1   #include <stdio.h>
2
3   #define MAX 100 //tamanho máximo do heap/vetor
4
```

Implementação do heap

Lembre-se que os índices assumem que o primeiro elemento está na posição 1 do vetor, e não na posição 0; apesar que podemos "ajustar" o código para não desperdiçar a posição 0.

```
 5   int pai(int i) {
 6       return i/2;
 7   }
 8
 9   int esq(int i) {
10       return 2*i;
11   }
12
13   int dir(int i) {
14       return 2*i+1;
15   }
16
```

Inserção no heap

```
17  void subir(int *heap, int i) {
18      int aux;
19      int p = pai(i);
20      if (p >= 1) { //se i não é a raiz pai(i) >= 1
21          if (heap[i] > heap[p]) {
22              //sobe no heap/vetor trocando pai e filho
23              aux = heap[i];
24              heap[i] = heap[p];
25              heap[p] = aux;
26             subir(heap, p);
27          }
28      }
29  }
30
```

Inserção no heap

```
31  void inserir(int *heap, int *n, int novoItem) {
32      //aumenta o tamanho do heap/vetor
33      *n = *n + 1;
34      heap[*n] = novoItem;
35      //sobe o último elemento do heap/vetor
36      subir(heap, *n);
37  }
38
```

Complexidade: O(log n)

Remoção do heap

```
39  void descer(int *heap, int n, int i) {
40      int aux;
41      //descobrir quem é o maior filho de i
42      int e = esq(i); //filho esquerdo
43      int d = dir(i); //filho direito
44      int maior = i; //inicialmente o maior é o pai
45      if ((e <= n) && (heap[e] > heap[i]))
46          maior = e; //filho da esquerda maior que o pai
47      if ((d <= n) && (heap[d] > heap[maior]))
48          maior = d; //filho da direita maior que o maior
49      if (maior != i) { //se o maior não é o pai
50          //desce trocando o pai com o maior filho
51          aux = heap[i];
52          heap[i] = heap[maior];
53          heap[maior] = aux;
54          descer(heap, n, maior);
55       }
56  }
57
```

Remoção do heap

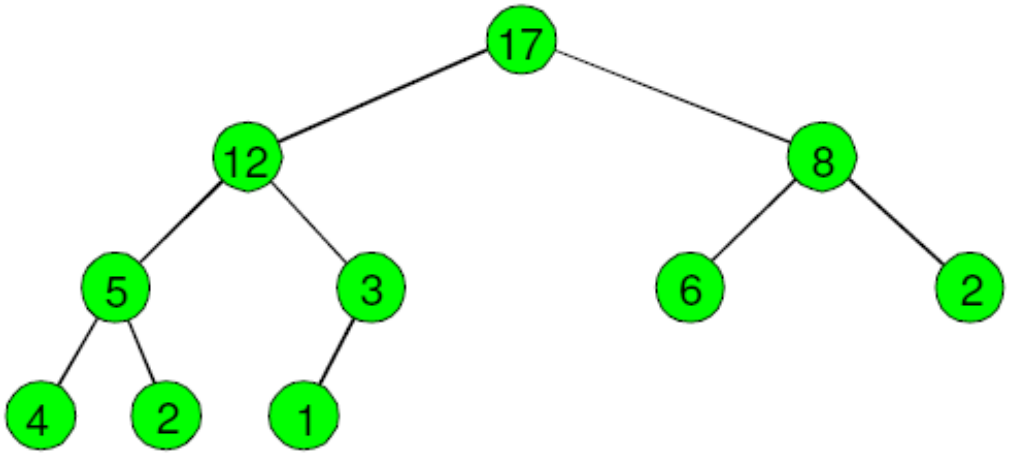
```
58  int removerMaximo(int *heap, int *n) {
59      int maximo = heap[1]; //maior elemento do heap/vetor
60      heap[1] = heap[*n]; //coloca o último elemento na raiz
61      //diminui o tamanho do heap/vetor
62      *n = *n - 1;
63      //desce o elemento na raiz do heap/vetor
64      descer(heap, *n, 1);
65      //retorna o maior elemento do heap/vetor
66      return maximo;
67  }
68
```

Complexidade: $O(\log n)$

Complexidade

- Inserção no heap
  O(log *n*)

- Remoção do heap
  O(log *n*)



Impressão do heap

```
69  void imprimirHeap(int *heap, int n, int i, int b) {
70      int j;
71      if (i > n) {
72          for (j = 0; j < b; j++) printf("-----");
73          printf("NULL\n");
74          return;
75      }
76      imprimirHeap(heap, n, dir(i), b + 1);
77      for (j = 0; j < b; j++) printf("-----");
78      printf("%i\n", heap[i]);
79      imprimirHeap(heap, n, esq(i), b + 1);
80  }
81
```

Complexidade: $O(n)$

Ordenação com heap

- Perceba que podemos ordenar os elementos em um heap sem dificuldades.

- Basta “extrair” (remover o máximo) os elementos um por um, ou seja, deslocar a raiz sempre para o final do heap.

Ordenação com heap

- O algoritmo de ordenação conhecido como heapsort, desenvolvido em 1964, explora exatamente esta ideia.

- Porém, precisamos fazer com que os elementos desordenados de um vetor passem à condição de heap.

Constrói Heap Descendo

```
82  void constroiHeapDescendo(int *heap, int n) {
83      int i;
84      int j = n/2;
85      for (i = j; i >= 1;i--)
86          descer(heap, n, i);
87  }
88
```

Complexidade: $O(n)$

Constrói Heap Subindo

```
89   void constroiHeapSubindo(int *heap, int n) {
90       int i;
91       for (i = 2; i <= n; i++)
92           subir(heap, i);
93   }
94
```

Complexidade: $O(n \log n)$

Heapsort

```
 95  void heapSort(int *heap, int n) {
 96      int i;
 97      int aux;
 98
 99      constroiHeapDescendo(heap, n); //constroiHeapSubindo(heap, n);
100      for (i = n; i > 1; i--){
101          //troca raiz (máximo) com o último elemento do heap
102          aux = heap[i];
103          heap[i] = heap[1];
104          heap[1] = aux;
105          //diminui o tamanho a ser considerado no heap
106          n = n - 1;
107          //desce com a raiz nesse novo heap de tamanho n-1
108          descer(heap, n, 1);
109       }
110  }
111
```

Complexidade do Heapsort

- constroiHeapDescendo
  O(*n*)

- constroiHeapSubindo
  O(*n*log *n*)

- heapSort
  O(*n*log *n*)

Implementação do heap

```c
int main(void) {
    int heap[MAX]; //vamos considerar um heap/vetor de inteiros
    int chave;
    int n = 0; //heap/vetor inicia com 0 elementos

    inserir(heap, &n, 17);
    inserir(heap, &n, 15);
    inserir(heap, &n, 12);
    inserir(heap, &n, 9);
    inserir(heap, &n, 10);
    inserir(heap, &n, 4);
    inserir(heap, &n, 1);
    inserir(heap, &n, 7);
    inserir(heap, &n, 3);
    inserir(heap, &n, 6);

    imprimirHeap(heap, n, 1, 0);
    printf("\nExtrair maior elemento do heap: %d\n\n", removerMaximo(heap, &n));

    imprimirHeap(heap, n, 1, 0);

    int vetor[11] = {100, 16, 22, 45, 99, 11, 27, 13, 34, 85, 76};

    heapSort(vetor, 11); //vetor[0] não é usado

    printf("\nVetor ordenado com Heapsort:\n");
    for (int i = 0; i <= 10; i++) printf("%d ", vetor[i]);

  return 0;
}
```

Exercício

Altere a implementação, em linguagem C, da estrutura de dados heap de tal forma que a raiz ocupe a posição 0, e não a posição 1.

Fim!